DCHORD: AN EFFICIENT AND ROBUST PEER TO PEER LOOKUP SYSTEM

Chul-Su Kim[1], Sanghwan Lee[2], Jae-Il Han[3], Yong-Joon Lee[4], Jong-Hyun Park[5]

[1,4,5]Electronics and Telecommunications Research Institute (ETRI),138, Gajeongno, Yuseong-gu, Daejeon, 305-700, South Korea.
[2, 3]School of Computer Science, Kookmin University, 861-1, Jeongneung-dong, Seongbuk-gu, Seoul, 136-702, South Korea.
Email : chulsu1@etri.re.kr[1], sanghwan@kookmin.ac.kr[2], jhan@kookmin.ac.kr[3], yjl@etri.re.kr[4], jhp@etri.re.kr[5]

ABSTRACT

Dynamic Hash Tables (DHTs) are distributed systems that maintain key-value pairs and provide efficient lookup services. Traditional DHTs usually rely on a random ID distribution of the keys to achieve such efficiency. Uniformly random hash functions are typically used to create uniformly random ID distributions from non-random key distributions. However, there are many cases where such random hash functions cannot be applied. For example, those systems that provide range queries over the keys cannot apply random hash functions on the keys, otherwise, the range query is very difficult to support. In this paper, we present a new lookup system called DChord, which does not depend on the randomness assumption to achieve its performance. To show the performance of the proposed system, we provide mathematical analysis and extensive simulation results in a highly non-random USN (Ubiquitous Sensor Network) metadata identifier space. To be specific, we show that DChord has high regularity in terms of in-degree and out-degree distributions. Thus, the system is robust against random node failures. We also show that query processing load is well balanced among nodes and the lookup speed is deterministic in such a way that the number of nodes to visit for a query is at most $log_2(N)$.

Keywords: *Lookup, USN, Deterministic, DHT, DChord*

1.0 INTRODUCTION

The success of peer-to-peer systems in industry and academia is clearly manifested by their high traffic volume in the Internet[1]. In particular, dynamic hash tables (DHTs), which maintain key-value pairs in a peer-to-peer network, have been extensively studied over the last several years. Chord ([2]), Pastry ([3]), CAN ([4]), and Leopard ([5]) are some examples of DHTs. Those systems provide scalable lookup performance so that as the number of nodes in the system increases, the performance degrades gracefully. DHTs usually convert an actual key (possible a string) into a number, called ID of the key (usually 128 bits) and maintain the value of the key in a node of which ID is near the ID of the key. One of the main ideas of these DHTs is that they often use random hash functions such as SHA-1 to distribute keys over a uniform ID space and exploit the uniform randomness to construct robust peer-to-peer networks that provide an efficient lookup speed and low maintenance overhead.

However, there are many cases where random hash functions cannot be applied for various reasons such as to support range queries. Order preserving hashing functions may solve the range query problem, but create other issues such as load unbalance among the nodes. Thus, if the uniform hashing function is not used, the resulting topology over the non-random original ID distribution becomes unbalanced in such a way that some nodes may have large number of neighbors and some nodes have small number of neighbors. Such an unbalanced topology is likely to be partitioned due to churning, where nodes join and leave frequently.

Motivated by this observation, in this paper, we propose a DHT scheme, called DChord, which does not rely on the randomness assumption. The main idea is to use node distribution rather than ID distribution in such a way that each node chooses about the same number of other nodes as its neighbors on the P2P overlay network. Furthermore, we provide an efficient and robust stabilization scheme by exploiting the overlay structure. We evaluate DChord in a highly non-random ubiquitous sensor network (USN) metadata ID space.

The contributions of this paper are summarized as follows.

- We propose an efficient peer to peer lookup system that performs better than the existing approaches for the non-random ID distributions. The system is rigorously defined in a mathematical formulation.
- We provide formal analysis on the performance of the proposed system.
- We present an efficient and robust stabilization scheme for DChord.
- We provide extensive simulation results that show the correctness of the formal analysis

The rest of the paper is organized as follows. Section 2.0 describes some background work on DHTs and some non-random ID distributions. The detailed operations of DChord are described in section 3.0. In section 4.0, we describe the performance evaluation via simulation. We conclude the paper in section 5.0.

2.0 BACKGROUND

2.1. Dynamic Hash Tables

P2P based lookup systems are often called Dynamic Hash Table (DHT). Early systems include Chord, CAN, and Pastry. In these systems, the nodes form a P2P network in a systematic way so that the lookup time does not increase much as the number of nodes increases. The keys are hashed into random numbers and the values are stored in a node based on the hashed random number. To find the node that contains the value of a key, clients follow the same systematic procedure as they store the key value pairs.

Among many DHT systems, we describe Chord in detail because DChord shares the spirit in many aspects. In Chord, each node randomly generates its own ID. The nodes form a ring based on their IDs. Specifically, the nodes are placed in a ring clockwise in the order of their IDs. Successor of a key is defined as the first node which has ID that is equal to or greater than the key. The key value pair is stored at the successor of the given key. To facilitate the lookup procedure, each node has 128 reference points (or fingers) to other nodes. Let *id* be the ID of a node. Then the fingers are successors of a series of IDs, $id+2^k$, where $k = 0, \ldots, 127$. In other words, each node has 128 fingers to other nodes in the system and the distribution of the fingers are in such a way that there are more finger nodes in the near and less finger nodes are in the far. Lookup operation is quite straightforward in that the next hop is one of the fingers, which is the nearest to the target node. Due to this specific form of the finger table and the next hop choosing procedure, the lookup operation takes only $O(log(N))$ hops where N is the total number of nodes. The assumption behind such a good performance is that the node IDs are random. In a uniformly random ID space, the 128 fingers are uniformly distributed among the nodes so that the resulting topology looks like a regular topology in such a way that the in-degrees and out-degrees are almost the same. For the non-uniform or non-numeric key spaces, Chord converts the original *non-random* key space into a *uniformly random* ID space by using a uniform hash function such as SHA-1 so that the structural properties of Chord can be achieved.

[6] discusses how to achieve the range queries in an efficient way. They propose to use hop space, which represents the hop distance among the nodes on the Chord-like ring, in such a way that the finger entries are chosen in an exponentially increasing order in the hop space. Their main contribution is to provide a systematic way to find the optimal finger distances with a given finger table size. To be specific, for a routing table (finger table) of size r, the fingers are chosen by the following formula.

$$d_i = round\left[n^{\left(\frac{i-1}{r}\right)} \right],$$

where d_i is the actually hop distance from the node. As the formula suggests, the distance exponentially increases as i increases so that there are more fingers are in the near distance. [6] shares the same sprit as DChord. However, they do not provide any efficient stabilization method, which is necessary for high churn and high ID distribution change situations.

One of the recent achievements for the range query is [7]. They present a platform for building a range-queriable system. They decouple the object and the node that contains the object in such a way that the object can freely move to other nodes while the efficient lookup is still provided. The main idea of how to achieve such performance is to make the joining node find its position that can reduce the load of highly loaded nodes so that the resulting load distribution is likely to be balanced. They propose many heuristics on how to find such a highly loaded node. DChord is different from their idea in that DChord provides deterministic approach to construct the topology rather than heuristic and probabilistic.

2.2 Non-Randomness Analysis

To understand how the non-random ID distributions can occur and to measure the non-randomness, in this sub section, we introduce a measure of non-randomness and present an example of highly non-random ID distributions. To quantitatively measure the non-randomness, we adopt the notion of ID *entropy* for the instances of IDs [8]. For a given number of bits, *b*, let X_i^b be an event where the number formed by the first *b* bits of an ID is *i*, $0 \le i < 2^b$, and $P(X_i^b)$ be the probability of X_i^b. We define ID-entropy, *H(ID)*, as the following

$$H(ID) = -\sum_{i=0}^{2^b-1} P(X_i^b)\log(P(X_i^b))$$

With this measure, H(ID), we quantify how much random a set of IDs are.

As an example of highly non-random ID distributions, we present a USN (Ubiquitous Sensor Network) metadata ID space. In the past several years, many researchers have been studying various issues in Ubiquitous Sensor Networks (USN) including operating system, programming language, fault tolerance, routing, and energy efficiency [9, 10, 11]. This active research is primarily due to the variety of the applications that the sensor networks can support [12]. However, since different applications require different services from the sensor networks, it is necessary to have a middleware that can provide a uniform interface to the applications regardless of what the physical sensor network is. This motivates the researchers to design a middleware service, so called the USN middleware, staying between the applications and the sensor networks [13]. Among many functionalities of the USN middleware, the directory service supports applications to access the metadata of various USN components such as sensor nodes and transducers. Due to the importance of the metadata format for the interoperability among heterogeneous USN systems, the USN metadata ID format is currently undergoing a standardization process in South Korea. One aspect of the USN metadata ID format is that it contains the location information (longitude and latitude) as prefixes, so it is expected that the IDs are highly localized when the possible locations are bounded.

To examine the non-randomness of the USN metadata ID space, we randomly generated 2^{16} IDs for Random, USN-Korea, and USN-USA. Random data set contains the IDs chosen randomly from 128 bit numbers. USN-Korea and USN-USA contain the sets of IDs chosen from the metadata IDs generated in the Korean Peninsula and the mainland USA respectively. Table 1 shows H(ID) of 2^{16} IDs over various numbers of bits. It is clear that the entropy of USN metadata IDs is low compared to that of the random IDs. In particular, when *b* is less than or equal to 5, the entropy of USN-Korea is 0, which means that the first 5 bits are the same for all the IDs. On the contrary, the entropy values of Random are close to the theoretical upper bound (UB). Since the area of the USA is much larger than that of Korea, it is expected that USN-USA show higher entropy values. Thus, the entropy clearly shows the degree of non-randomness.

Table 1 H(ID)

b	Random	USN-Kor	USN-USA	UB
4	3.9998	0	1.8842	4
5	4.9996	0	2.6759	5
6	5.9993	0.9063	3.6336	6
7	6.9985	1.5822	4.5766	7
8	7.9973	2.5740	5.5618	8

3.0 DCHORD LOOKUP SYSTEM

In this section, we present our design of DChord as an instance of Dynamic Hash Tables. To be specific, we first describe the finger table and then we describe the basic operations that DChord provides. We also provide formal analysis on the performance of DChord in query processing and maintaining the structure.

3.1. Overview

In DChord system, the nodes form a ring structure among themselves based on their IDs. Each node has a unique id in the ID space of $[0,2^m)$. The node with a higher ID comes later in clockwise direction as can be seen in Fig.1

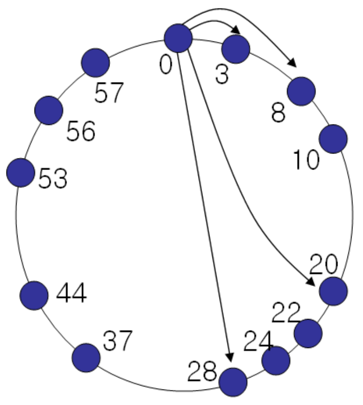

Fig.1. Example of node distribution in DChord.

In Fig.1, the nodes are placed on a ring in clockwise order and the ID space is [0, 64), i.e., *m*=6. The node that comes right before a node is called the *predecessor* and the node that comes right after a node is called the *successor* of the node. Each node maintains the contact information of the predecessor and the successor. In addition to that, each node maintains a finger table with the following structure. To provide the definition of the finger table, we first define some notations.

- Let $[0,2^m)$ be the ID space of *m* bits.
- Let *N* be the number of nodes in the system.
- Let $k = \lceil \log(N) \rceil$.
- Let *n(i)* be the i^{th} node in the order of the IDs. As a result, node *n(i+1)* is the successor of *n(i)* and *n(i-1)* is the predecessor of *n(i)*. One thing to note is that *i+1* and *i-1* are all mod *N* operations. For example, in Fig.1, N=14 and *n(13)* is node 57. Then, *n(13+1)=n(14* mod *14) = n(0)*, which is actually node 0.

Besides the successor and the predecessor, each node chooses the first node, second node, the fourth node, eighth node, and so on, from itself in a clockwise direction and maintains the contact information of the chosen nodes in a data structure called Finger table. In general, each node chooses the set of nodes that are apart from the node in the 2^j-th positions in a clockwise direction, where $0 \le j < \log_2(N)$. The finger table, $F(n(i))$, of node *n(i)* is rigorously defined as follows.

$$F(n(i)) = \{(S_{n(i),j}, J_{n(i),j})\} \quad for\ 0 \le j \le k\ ,$$

where $S_{n(i),j}$ is a range and $J_{n(i),j}$ is a node (called a jump node), which is the node to which *n(i)* sends queries of an ID in the corresponding range. In other words, the finger table is a set of range-jump node pairs. The 0th range,

$$S_{n(i),0} = [n(i), n(i+1)),$$

is the range that the node *n(i)* covers, which is basically the range from its own ID to just before the ID of the successor. In short, each node maintains the key-value pairs whose key falls between the ID of the node and that of the successor of the node.

The finger table entries must satisfy the following three constraints to efficiently run the lookup operation.

1) Coverage Constraint

$$\bigcup_{0 \le j \le k} S_{n(i),j} = [0,2^n)$$

2) Jump node constraint

$$J_{n(i),j} = \begin{cases} n(i), & j = 0 \\ n(i+2^{j-1} \bmod N), & 1 \le j \le k \end{cases}$$

3) Range constraint

$$S_{n(i),j} = \begin{cases} [n(i), n(i+1 \bmod N)), & for\ j=0 \\ \bigcup_{2^{j-1} \le l < 2^j} S_{n(i+l \bmod N),0} & for\ 1 \le j < k \\ [n(i+2^{k-1} \bmod N), n(i)), & for\ j=k \end{cases}$$

The coverage constraint is quite straightforward because if the union of the ranges in the finger table does not cover all the ID space, then there can certainly be some area for which the node cannot forward the query message. The jump node constraint is the essence of DChord. In short, the set of jump nodes is the set of nodes that are apart from the node in 2^j th position where $0 \le j < k$. For example, in Fig.1, *n(0)* is the node 0, *n(1)* is node 3, *n(2)* is node 8, and so on. Then, node 0 has the set of jump nodes, {3, 8, 20, 28}, because node 3 is the first node after node 0, node 8 is the second node, node 20 is the fourth node, and node 28 is the eighth node from node 0. After we determine the jump node, the range constraint is straightforward. It means that the range should be the union of ranges that each node covers from the jump node to the predecessor of the next jump node.

For example, $S_{n(0),3} = \bigcup_{2^2 \le l < 2^3} S_{n(i+l),0} = S_{n(4),0} + S_{n(5),0} + S_{n(6),0} + S_{n(7),0}$

In Fig.1, it would be the union of areas of node 20, 22, 24, and 25, i.e., the set of nodes from the fourth node to the predecessor of the eighth node. Table 2 shows the finger table of node 0 and 20. It should be noted that $k = 4$ because there are 14 nodes ($N = 14$).

Table 2. Finger tables of node 0 and 20

j	*n(i)* = 0		*n(i)* = 20	
	$S_{0,j}$	$J_{0,j}$	$S_{20,j}$	$J_{20,j}$
0	[0, 3)	0	[20,22)	20
1	[3, 8)	3	[22,24)	22
2	[8, 20)	8	[24,28)	24
3	[20, 28)	20	[28,56)	28
4	[28, 0)	28	[56,20)	56

3.2. Location Registration, Withdrawal, and Lookup

We now describe how DChord implements the three DHT operations: registration, withdrawal, and lookup of the key-value pairs. We assume that the key space is the same as the ID space. In other words, the keys are in $[0, 2^m)$.

Let a *home node* be the node that covers the given key, i.e., the node *n(i)* where $key \in S_{n(i),0}$.

Basically, DChord stores a key-value pair in the home node of the key. Therefore, all the three operations require a search for the home node with the given key.

When a node *n(i)* receives a request with a key *k*, *n(i)* searches the range that contains *k* in the finger table. If the *k* is in the 0th range, *k* belongs to the node and the node is the home node. Otherwise, it forwards the request to the jump node of the range that contains *k* until the home node is found. This home node searching procedure can be implemented recursively or iteratively. This operation is guaranteed to finish in a finite number of forwarding. Once the home node is found, the remaining operation is straightforward. For the registration operation, the key-value pair is stored at the home node. For withdrawal operation, the key-value pair is revoked from the home node. For query operation, the key-value pair is returned to the query initiating node.

In short, in the message forwarding procedure, the node that receives a message with a key, *k*, first checks whether it covers the key. If so, the node is the destination. Otherwise, the node chooses the finger table entry that has the largest ID but is less than or equal to *k*. Then, it forwards the request message to the chosen node.

This process repeats until the message arrives at the destination that covers the key.

The performance of the three operations depends on that of the home node search operation. Let *path length* be the number of nodes that needs to be visited to find the home node. If the search operation is recursive, the path length is the number of hops starting from the search initiating node to the home node. The following theorem shows the upper bound of the path lengths.

Theorem 1 : The path length of the home node searching operation is at most $log_2(N)$ under the three Constraints, where N is the number of nodes in the DChord system.

Proof. Let s be the starting node and t be the target home node. If $s = t$, then the search is done. If $s \neq t$, then s finds a jump node whose range contains the key. Let j be the position of the jump node in the finger table. Since $s \neq t$, j is not 0. By the Jump node constraint, the home node should be one of the 2^{j-1} nodes starting from the chosen jump node. Furthermore, when the next hop chooses another jump node, the position of the chosen jump node in the finger table is less than j because the number of nodes that precedes the l^{h} jump node is exactly 2^{l-1}. Therefore, the position of the chosen jump node is strictly decreasing at each hop. This procedure is repeated until the home node is found. So the path length is at most j, which is at most $log_2(N)$. □

3.3. Node Join, Leave, and Stabilization

DChord system is a dynamic network where nodes can join and leave. The node joining procedure in DChord system is defined as follows. When a node x joins the DChord network, it first finds the home node, h, of its own ID. Let s be the successor of h. Then, x puts itself between h and s. In other words, h becomes the predecessor of x and s becomes the successor of x. So h sets x as its successor and s sets x as its predecessor. Finally, x copies the finger table of h and uses it as its own finger table. The node leaving procedure is similar. When a node x leaves the network, the predecessor of x sets the successor of x as its successor and the successor of x sets the predecessor of x as its predecessor. The predecessor of x removes x from its finger table.

The joining and leaving procedures defined as above are very simple. However, every time a node joins or leaves, the finger tables of the nodes violate the three constraints. Furthermore, it is not scalable to adjust all the finger tables at every join and leave operation because the number of nodes that need to adjust the finger table is about half of the nodes in the system. So, we design that DChord maintains only the predecessor and the successor information to be correct. It should be noted that the forwarding is still possible even if the finger table does not satisfy the finger table requirement.

However, as the nodes join and leave, the entries of the finger table may not refer to the nodes of the 2^{j}-th positions if they are not updated. Thus, in DChord the finger tables are updated periodically. This procedure is called *stabilization*. The stabilization procedure in DChord depends on the following property.

$$\begin{aligned} J_{n(i),j} &= n(i+2^{j-1}) \\ &= n((i+2^{j-2})+2^{j-2}) = J_{J_{n(i),j-1},j-1} \end{aligned}$$

The above formula can be rephrased as follows. When a node x wants to adjust the j^{th} jump node, it contacts the $j\text{-}1^{th}$ jump node (say t) to get the $j\text{-}1^{th}$ jump node (say z) of t because z is actually j^{th} jump node of x. For example, in Table 2, the 3rd jump node of node 0 is node 20, and the 3rd jump node of node 20 is node 28. So, the 4th jump node of node 0 is node 28. In DChord system, at each stabilization period, each node adjusts the jump nodes from the lowest position to the highest. So the node first adjusts the 2nd jump node (0th and 1st jump nodes are itself and the successor node, so they are always correct.) Then, it adjusts the 3rd, and 4th, and so forth.
The stabilization procedure is depicted in Table 3. *Finger(i)* represents the *i*-th finger entry. *Finger(0)* is the successor node.

Table 3. Stabilization procedure of a node.

Stabilization()

for (i = 0; Finger(i) doesn't wrap around; i++)
    Send request msg to Finger (i) for its i-th finger entry
    Receive response msg from Finger (i)
    Set Finger ($i+1$) with the response

In summary, this property can be simply described as follows.

The j-th finger entry of a node is the (j-1)-th finger entry of the (j-1)-th finger entry of the node.

A synchronized stabilization is the one that all the nodes run the finger table update at the same phase. To be specific, the Finger (i+1) updates for all the nodes are initiated after all the Finger (i) updates of all the nodes are finished so that the finger entries received from other nodes are all valid. However, synchronization may not be easy to achieve among millions of nodes. Instead of running the stabilization for all the finger entries at one time, we may spread the updates of the finger entries over time so that we have enough time for other nodes update lower finger entries. Furthermore, to prevent any problem of synchronized operations, we may introduce a randomized update periods. More sophisticated stabilization process will be investigated in the future work. We also consider non periodic stabilization process. For example, a node starts the stabilization process whenever a new node joins before or after itself.

The number of nodes for a node to contact at each stabilization period is $log_2(N)$. Therefore, ideally, constructing finger tables of all the nodes from empty finger tables takes only $N\ log_2(N)$ messages. In Chord, it takes $O(\log^2(N))$ for each join and leave so that the whole finger table construction takes $O(N\log^2(N))$, which is larger than that of DChord by a factor of O(log(N)). However, during the stabilization period, the path lengths can be longer because the three constraints can be violated in between the stabilization periods. The following theorem shows that the penalty (the number of extra hops) in the path lengths in between the stabilization periods is small.

Theorem 2. The average path length penalty after k nodes join to a N-node stabilized DChord system without stabilization is $\frac{2k}{N+k}$ hops for $k \le N$ with high probability.

Proof : Before the additional k nodes join, the finger tables satisfy the three Constraints and there was no stabilization process until the k nodes join. There are N slots to which the k nodes can be inserted. Therefore, the average number of new nodes between a pair of consecutive original nodes is

$$\sum_{i=0}^{k} i\left(\frac{1}{N}\right)^{i} \le 1$$

So we can assume that there is at most one new node between any two old nodes. We assume that the queries are uniformly distributed over all the nodes.
There are 4 cases based on whether the query originator and the target nodes are new or old. Since the newly joined node adjusts only predecessor and successor and there is at most one node between any two old nodes, the penalty for each case is as follows.

- From an original node to a newly joined node : 1 hop penalty
  The query will be forwarded to the predecessor of the newly joined node and then passed to the target.
- From an original node to an original node : 0 hop penalty
  The original nodes satisfy the three Constraints, so no penalty occurs.
- From a newly joined node to an original node : 1 hop penalty
  Since the newly joined node has only successor and predecessor, the query will be forwarded to the successor and then forwarded to target node.
- From a newly joined node to a newly joined node : 2 hop penalty
  The queries are forwarded to the successor of the query originator and to the predecessor of the target node and then to the target node. So the penalty is 2 hops, the first and the last.
  Let $AP(n,k)$ be the average penalty when k nodes join to a N node DChord system.

$$AP(n,k) = 1\cdot\frac{Nk}{(N+k)(N+k-1)} + 0\cdot\frac{N(N-1)}{(N+k)(N+k-1)}$$
$$+1\cdot\frac{kN}{(N+k)(N+k-1)} + 2\cdot\frac{k(k-1)}{(N+k)(N+k-1)}$$
$$=\frac{2k}{N+k} \qquad \square$$

4.0 EVALUATION

4.1. Simulation Methodology

We now evaluate the performance of DChord with several performance metrics. The simulator is implemented in Java with the typical discrete event simulation methodology. Each event (node join, leave, lookup message transmission, etc) has the creation time and a priority queue maintains all the events that are not processed yet.

The performance metrics are *path length*, *in-degree, out-degree*, and *query processing count*. The in-degree of a node is the number of other nodes that have finger information to this node. The out-degree of a node is the number of *distinct* nodes that the node has in its finger table. So the in-degree and out-degree represent the robustness of the structure in such a way that the larger the degrees are the harder the system breaks down. The query processing count shows how many queries each node processes. The performance metrics are computed for DChord and compared with that of Chord.

We use two kinds of ID spaces for the simulations. One is the Random ID space and the other is the Non-Random ID space. The Random ID space is the one where all node IDs are generated uniformly randomly from the 128 bit pattern. In non-Random ID space, we use the USN-Korea data set, which has been discussed in section 2.0. The reason to use Random ID space is because we want to show that DChord is also good for Random ID space.

4.2. Load Balancing

In DChord and Chord, each node should maintain the finger table, which contains the contact information to other nodes. The size of the finger table is largely affected by the distribution of the node IDs. For a random node IDs, Chord can only maintain O(log(N)) finger nodes. However, when the ID space is non-Random as that of USN metadata, O(log(N)) may not be achieved. To see the effect, we compute the in-degree and out-degree of each node in each scheme.

Before we look into the Non-Random ID space, we first look at the Random ID space case. Fig. 2 shows the distribution of in-degrees in Chord and DChord in the Random ID space. Specifically the minimum, maximum and average in-degrees are shown in the figure for each node size. As the number of nodes increases, the in-degree increases. One distinct difference between Chord and DChord is that DChord shows very stable distribution, i.e., the min, max, and average are the same. However, in Chord there are some nodes that have very small in-degree and some nodes that have very large in-degree. When a node has very large in-degree, it is likely to receive more messages from other nodes so that the message processing load increases for such nodes. However, DChord shows no variation in the in-degree, so the load in each node is well distributed.

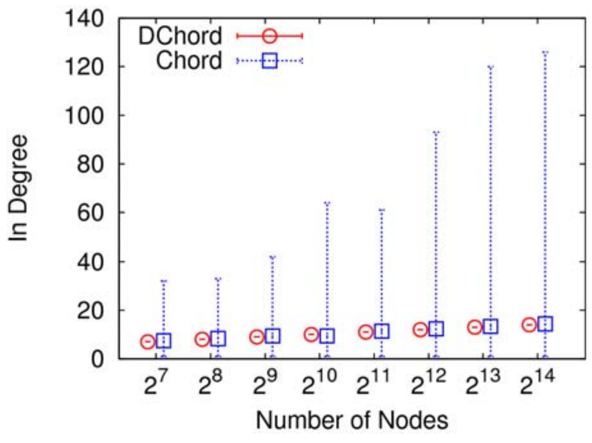

Fig. 2. In-degree with Random ID Space

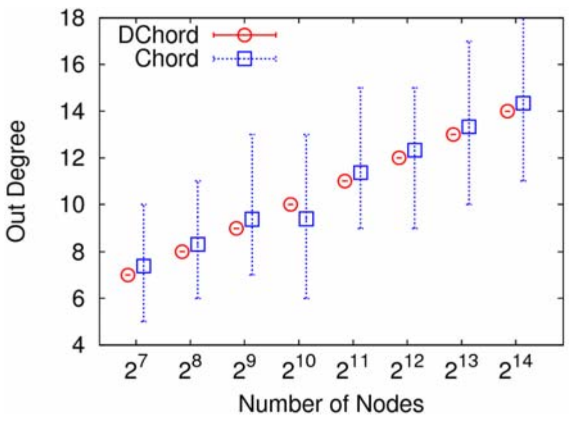

Fig. 3. Out degree with Random ID space

Similarly we compute the distribution of out-degree with Random ID space. Fig. 3 shows the min, max, average in-degree in Chord and DChord in Random ID space. As can be seen in the figure, the out-degree increases as the number of nodes increases. However, Chord shows high variation of the out-degree, which means that the nodes with small out-degree is vulnerable to node failures because when all the nodes in the finger table fail, the node is actually disconnected from the system. On the other hand, DChord shows no variation in the out-degree, so that the robustness of each node is about the same. Another impact of the in-degree and out-degree distribution is on the amount of periodic message exchange to check whether the neighbors are alive. Since the degrees are the same for all the nodes in DChord, the amount of messages that each node processes is about the same. However, in Chord, the message processing loads are very different, which incurs load unbalance to the system.

The degree distribution shows more severe result in Non-Random ID space. Fig. 4 shows the min, max, average in-degree of Chord and DChord with Non-Random ID space. The most prominent result is that there is a node that has the in-degree which is the same as that of the number of nodes in Chord. In other words, all the other nodes have finger information to this node. This means that when the node with the largest in-degree fails, all the other nodes lose one contact point to the system. Furthermore, this node behaves like a central server in the system. So a large number of messages are forwarded to this node, which cause unbalanced load distribution to the system. Similar result is shown in Fig. 5. Fig. 5 shows the min, max, and average out-degree of Chord and DChord with Non-Random ID space. For each node count, the min out-degree is 1 in Chord. This means that there exists a node that has only one *distinct* node in the finger table. So when the only finger node fails, the node is excluded from the system, i.e., it cannot run any lookup operation because there is no contact point in the system that the node can use. However, in DChord, the out-degrees have no variation among nodes, which is the most important advantage over Chord.

The even distribution of degrees in DChord makes the system more balanced in terms of the number of query messages that each node processes. Fig. 6 shows the min, max, average query messages that each node receives over the total number of query messages. The number of query messages that each node receives increases as the total number of queries increases. In Chord, the max count is much higher than that of DChord. The distribution is wider compared to that of DChord. Fig. 6 clearly shows that DChord is more balanced in terms of the number of queries that each node processes.

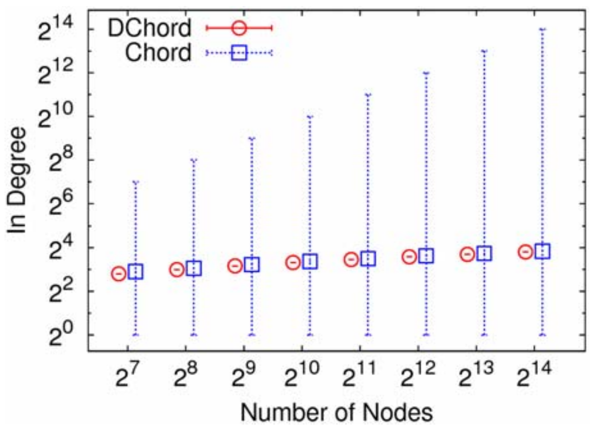

Fig. 4. In-degree with Non-Random ID space

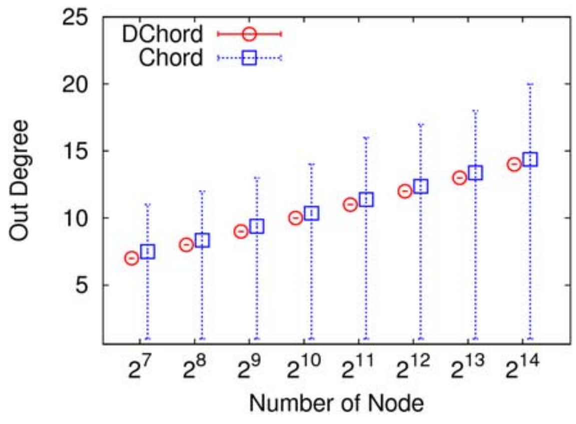

Fig. 5. Out-degree in Non-Random ID space

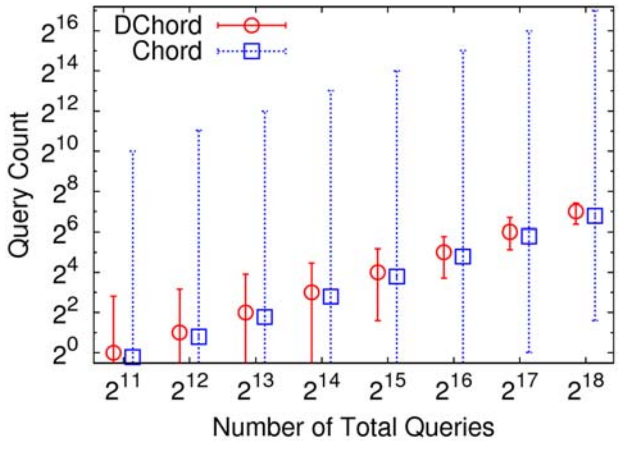

Fig. 6. Min, Max, Average number of queries messages over total query count

4.3. Lookup Overhead

Peer to Peer structure has some disadvantages in the lookup speed compared to that of a centralized system because in the centralized system, only one transmission of a query message to the centralized server is enough to get the location information. However, in peer to peer system, the query message is relayed through many peers to reach the destination.

To see the overhead of the lookup operation, we compute the path length of the queries. Fig. 7 shows the 1st percentile, 99th percentile, and the average of path lengths over the number of nodes with the Non-Random ID space. As the number of nodes increases, the path lengths increase. As can be seen in the figure, the average path lengths are much less than *$log_2(N)$*. It should be noted that the unbalanced structure of Chord in Non-Random ID space can actually decrease the path lengths because the node with the large *in-degree* behaves as a centralized node so that most of the query messages are first forwarded to the node, and then to other nodes.

A more precise distribution of the path lengths is given in Fig. 8. We compute the percentage of queries that has each path length with the Non-Random ID space. The total number of nodes is 2^{14}. The most queries have path length of 7, which is the half of *$log_2(2^{14})$*. This result is similar in other node counts. So we would expect that the path length does not cause large overheads on the response time.

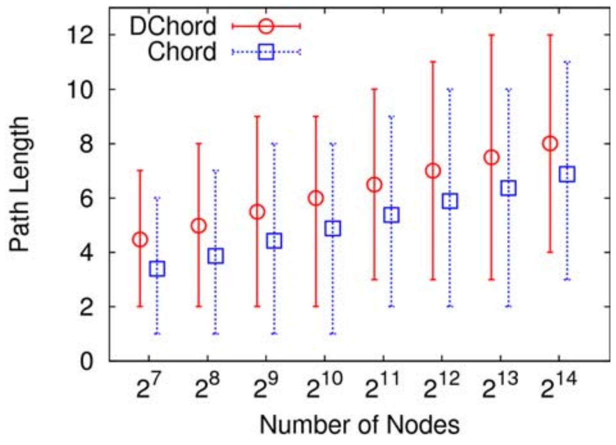

Fig. 7. Path Lengths of Query Messages

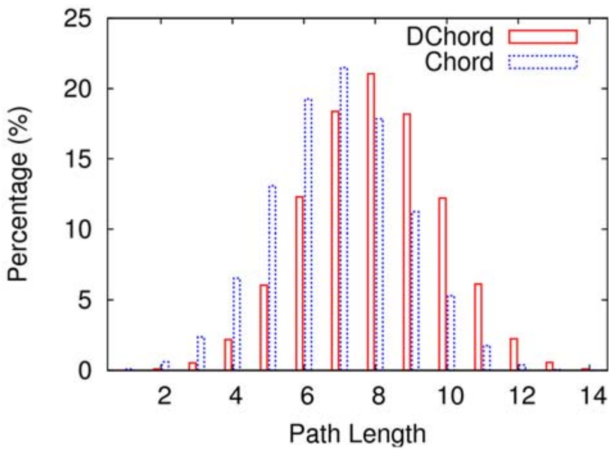

Fig. 8. Percentage of Queries over Path Lengths

4.4. Unbalanced Chord Topology for non-Random ID distributions.

In Chord system, if hashing is not used, the Chord network is vulnerable to node failures due to the non-random node ID distribution. At the same time, load balancing cannot be achieved. For example, in Fig. 9, which shows a Chord network, node 36 is the single point of failure because most of the other nodes have a finger table entry to this node. To be precise, node 0 has a finger table whose entries are all node 36 because there is a big empty range between node 0 and node 36. So if node 36 failed, then node 0 cannot forward queries because there is no other jump node in the table other than node 36.

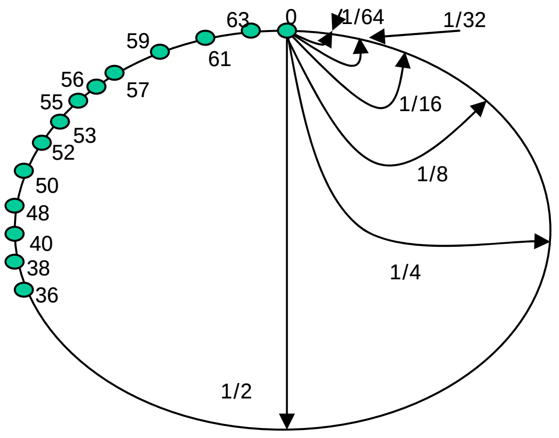

Fig. 9. Vulnerability of Chord System for non-Random ID and Node space

5.0 CONCLUSION

In this paper, we propose a lookup system that is suitable for non-random ID space. The non-random ID space makes it difficult to use an existing P2P lookup system that is based on the uniformly random ID assumption. The proposed system, DChord, does not depend on the randomness assumption. It can construct a uniform and load balanced lookup structure, so that the in-degree and out-degrees are the same for all nodes. We also show that the path lengths are less than or equal to $log_2(N)$. Furthermore, we present an efficient stabilization procedure so that the maintenance cost of the topology has been reduced by a factor of log(N) compared to other DHT systems such as Chord. We conclude that DChord is a viable solution for the systems that have non-random ID distribution such as USN metadata. On the other hand, we have not looked at the effect of actual response time for the lookup operations. Since one hop in DChord may take a very large one way delay, the response time may not be proportional to the path lengths. On the other hand, the IDs are based on geographical location, so there is a chance to alleviate the difference between the path length and the actual delay by exploiting the geographical location. We plan to investigate this issue further in the future work.

ACKNOWLEDGEMENT

This work was partly supported by the IT R&D program of MIC/IITA [2006-S022-02, Development of USN Middleware Platform Tech] and research program 2009 of Kookmin University in Korea. **Sanghwan Lee** is the corresponding author of this paper.

REFERENCES

[1] Myung-Sup Kim, Young J. Won, and James Won-Ki Hong, “Application-Level Traffic Monitoring and an Analysis on IP Networks ,” *ETRI Journal*, Vol. 27, No.1, Feb. 2005, pp. 22-42.

[2] I. Stoica, et al., “Chord: A scalable peer-to-peer lookup service for Internet applications,” in *Proceedings of ACM SIGCOMM 2001*, San Diego, CA, Aug. 2001, pp. 149-160.

[3] A. Rowstron and P. Drushel, “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems,” in *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, Nov. 2001, pp. 329-350.

[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content-addressable network,” in *Proceedings of ACM SIGCOMM 2001*, San Diego, CA, Aug. 2001, pp. 161-172.

[5] Y. Yu, S. Lee, and Z.-L. Zhang, “Leopard: A locality-aware peer-to-peer system with no hot spot,” in *Proceedings of the 4th IFIP Networking Conference (Networking’05)*, Waterloo, Canada, May 2005, pp. 27-39.

[6] F. Klemm, et al., “On routing in distributed hash tables,” in *Proceedings of the Seventh IEEE International Conference on Peer-to-Peer Computing*, Sept. 2007, pp. 113-120.

[7] Yuh-Jzer Joung, “Approaching neighbor proximity and load balance for range query in P2P networks,” *Computer Networks*, Vol. 52, No. 7, May 2008, pp. 1451-1472.

[8] K. Sayood, *Introduction to Data Compression*, 2nd ed. Morgan Kaufmann, pp. 13-22, 2000.

[9] M. Weiser, “Some computer science issues in ubiquitous computing,” *Communications of the ACM*, Vol. 36, No. 7, July 1993, pp. 75–84.

[10] C. Intanagonwiwat, R. Govindan, and D. Estrin, “Directed diffusion: a scalable and robust communication paradigm for sensor networks,” in *Proceedings of the International Conference on Mobile Computing and Networking (ACM Mobicom)*, Boston, MA, Aug. 2000, pp. 56-67.

[11] E.-S. Jung and N. H. Vaidya, “A power control mac protocol for ad hoc networks,” in *Proceedings of the International Conference on Mobile Computing and Networking (ACM Mobicom)*, Atlanta, GA, 2002, pp. 36–47.

[12] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “A survey on sensor networks,” *IEEE Communications Magazine*, Vol. 40, No. 8, Aug. 2002, pp. 102–114.

[13] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo, “Middleware to support sensor network applications,” *IEEE Networks*, Vol. 18, No. 1, Jan. 2004, pp. 6-14.

BIOGRAPHY

Chul-Su Kim is a staff in ETRI.

Sanghwan Lee received the Ph.D. degree in computer science and engineering from the University of Minnesota, Minneapolis, in 2005. Dr. Lee is currently an Assistant Professor at Kookmin University, Seoul, Korea. From June 2005 to February 2006, he worked at IBM T.J. Watson Research Center, Hawthorne, NY. His main research interest is theory and services in the Internet. He is currently applying the embedding schemes to a live peer to peer multimedia streaming system to see the real impact of the accuracy on the application

Jae-Il Han received the Ph.D. degree in computer and information science from Syracuse University in 1992. Dr. Han is a professor at the School of Computer Science in Kookmin University. His research interests include distributed systems, middleware, ubiquitous sensor network, embedded systems, services computing and security.

Yong-Joon Lee received the MS degree from Yonsei University, Korea, in 1988, and the PhD degree in computer science from Chungbuk National University, Korea, in 2001. He is currently working at ETRI, Korea. His major research interests include RFID and sensor data management, sensor data mining, context awareness in ubiquitous computing environment, and the development of sensor network middleware.

Jong-Hyun Park received the BS and MS degrees in the Departments of Space Science and Astronomy & Meteorology in KyungHee and Yonsei Universities, Seoul, Korea, in 1989 and 1991, respectively. He received the PhD degree in environmental engineering from Chiba University, Japan, in 2000. Since 1991, he has been with ETRI, where he is currently a Director of the Telematics Research Group. His research interests include location-based services, telematics, WSN (wireless sensor network) middleware, and positioning systems using a global navigation satellite system.